

Chapter 2

Formal Design Flows for Embedded IoT Hardware

Michael Dossis

2.1 Introduction

Embedded systems are usually relatively small/medium computing platforms that are self-sufficient. Such systems consist of all the software and hardware components which are “embedded” inside the system so that complete applications can be implemented and run without the aid of other external components or resources. Usually, embedded systems are found in portable computing products such as PDAs, mobile, and smart phones as well as GPS receivers. Nevertheless, larger systems such as microwave ovens and vehicle electronics contain embedded systems. Nevertheless, here embedded systems are considered that can communicate with each other by means of a wired or wireless communication protocol, such as Zigbee, IEEE.802.11 standard, or any of its derivatives. Therefore, special attention is paid here to embedded Internet-of-Things (IoT) hardware, its design methodology, and implementation requirements.

An embedded platform with such communication features can be thought of as a system that contains one or more general-purpose microprocessor or microprocessor core, a number of standard peripherals, along with a number of customized, special function co-processors, accelerators or special function engines on the same electronic board or integrated inside the same system-on-chip (SoC). Already a number of IoT manufacturers include hardware encryption and cryptography blocks in order to increase the security capability of their devices. Normally, specific blocks such as encryption/decryption and video processing engines can be attached on the embedded system’s bus and offer hardware accelerated functionality to the IoT module. An embedded and portable system that includes all of the above hardware/software modules can be thought of as a complete, standalone IoT node.

M. Dossis (✉)

TEI of Western Macedonia, Kastoria Campus, Fourka Area, Kastoria, 52100 Greece

e-mail: dossis@kastoria.teikoz.gr

Currently, such embedded systems are implemented using advanced field-programmable gate arrays (FPGAs) or other types of programmable logic devices (PLDs). Alternatively an embedded IoT node can be implemented with ASIC + SoC logic plus a microcontroller core, but this is financially viable only for large sale volumes. For smaller volumes, or at least for designing the prototype, FPGA logic is the best solution in terms of price/functionality yield. Nowadays, FPGAs offer a very large integrated area, circuit performance, and low power capability. FPGA implementations can be seamlessly and rapidly prototyped, and the FPGA can be easily reconfigured when design updates or bug fixes are released.

Advances on chip integration technology have made possible such a complexity of embedded and custom integrated circuits (ICs) that often their spec-to-product time exceeds even their product lifetime in the market. This is particularly true for commercial embedded electronics with product lifetimes compressed to less than a quarter of a year sometimes. Of course this is expected to be true for current and future IoT devices as well. This, in combination with the high design cost and development effort of such products, they often even miss their market windows, generating in turn, competitive disadvantages for the producing industries. The current engineering practice for the development of such systems includes to a large extent methodologies which are semi-manual, add hoc, segmented, not-fully integrated, empirical, incompatible from one level of the design flow to the next, and with a lot of design iterations caused by the discovery of functional and timing bugs, as well as specification to implementation mismatches late in the development flow.

This chapter reviews previous and existing work of HLS methodologies for embedded systems. It also discusses the usability and benefits using the prototype hardware compilation system which was developed by the author. Section 2.2 discusses related work and bibliography. Section 2.3 presents HLS problems related to the low energy consumption which is particularly interesting for embedded system design. The C-Cubed hardware compilation design flow is explained in Sect. 2.4. Section 2.5 explains the formal nature of the prototype compiler's formal logic inference rules. In Sect. 2.6 the mechanism of the formal high-level synthesis transformations of the back-end compiler is presented. Section 2.7 outlines the structure and logic of the PARCS optimizing scheduler which is part of the back-end compiler rules. Section 2.8 explains the options for generating target FSM + datapath micro-architectures and the communication of the accelerators with their computing environment. Section 2.9 outlines the execution environment for the generated hardware modules as accelerators. Section 2.10 discusses experiments with synthesizing hardware within the C-Cubed framework such as formal synthesis and verification, and Sect. 2.11 draws useful conclusions and proposes future work.

2.2 Background and Existing Work

2.2.1 *High-Level and Logic Synthesis*

All of the issues reported in the introduction and elsewhere have motivated industry and academia to invest in automated, integrated, and formal design automation methodologies and tools for the design and development of embedded ICs and systems. These methods are based on the idea of transforming a textual software-program-code-like description into a netlist of logic gates. Nowadays, a higher level of code abstraction of hardware description formats, such as VHDL and Verilog, C, SystemC, and ADA, is pursued as input to automated high-level E-CAD tools. Methodologies such as high-level synthesis (HLS) and electronic system level (ESL) design employ established techniques, borrowed from the computer language program compilers and established E-CAD tools as well as new algorithms such as advanced operation scheduling, loop unrolling, and code motion heuristics.

Currently, the design of digital systems involves programming of the circuit's functionality at the register-transfer level (RTL) level in hardware description languages such as VHDL and Verilog. However, for custom designs that are larger than a hundred thousand logic gates, the use of RTL code for specification and design usually results into years of design flow iterations and verification simulations. Combined with the short lifetime of electronic products in the market, this constitutes a great problem for the industry. Therefore, there has been a pressing need for more abstracted input formats such as C, C++, and ADA. However, the programming style of the (hardware/software) specification code has an unavoidable impact on the quality of the synthesized system. This is deteriorated by models with hierarchical blocks, subprogram calls, as well as nested control constructs (e.g., if-then-else and while loops). The complexity of the transformations that are required for the synthesis tasks (compilation, algorithmic transformations, scheduling, allocation and binding) of such high-level code models increases at an exponential rate, for a linear increase in the design size.

During HLS, the input code (such as ANSI-C or ADA) is first transformed into a control/dataflow graph (CDFG) by a front-end compilation stage. Then various optimizing synthesis transformations are applied on the CDFG to generate the final implementation. The most important HLS tasks of this process are scheduling, allocation and binding. Scheduling makes an as-much-as-possible optimal order of the operations in a number of control steps or states, parallelizing as many operations as possible, so as to achieve shorter execution times of the generated implementation. Allocation and binding assign operations onto functional units, and variables and data structures onto registers, wires, or memory positions, available from an implementation library.

A number of commercial HLS tools impose their own extensions or restrictions on the programming language code that they accept as input, as well as various shortcuts and heuristics on the HLS tasks that they execute. Such tools are the CatapultC by Mentor Graphics, the Cynthesizer by Forte Design Systems, the

Impulse CoDeveloper by Impulse Accelerated Technologies, the Synfony HLS by Synopsys, the C-to-silicon by Cadence, the C to Verilog Compiler by C-to-Verilog, the AutoPilot by AutoESL, the PICO by Synfora, and the CyberWorkBench by NEC System Technologies Ltd. The analysis of these tools is not the purpose of this work; however, they are tuned towards linear, dataflow dominated (e.g., stream-based) applications, such as pipelined digital signal processing (DSP) and image filtering.

In order to guarantee that the produced circuit implementations are correct-by-construction (with regard to the input specification functionality) it is mandated that the HLS tool's transformation tasks (e.g., within the scheduler) are based on formal techniques. In this way, repetitive execution of the design flow verification process will be avoided and important time will be saved within the development project. Correct-by-construction also means that by definition of the formal process, the functionality of the implementation matches the functionality of the behavioral specification model (the source code). In this way, the design will need to be verified only at the behavioral level, without spending very long and time-consuming simulations of the generated RTL, or even worse of the netlists generated by a subsequent commercial RTL synthesizer.

Behavioral verification (at the source code level) is orders of magnitude faster than RTL or even more than gate-netlist simulations. Releasing an embedded IoT product with bugs can be very expensive, when considering the cost of field upgrades, recalls, and repairs. Another thing, less measurable, but very important as well, is the damage done to the industry's reputation and the consequent loss of customer trust. However, many embedded products are indeed released without all the testing that is necessary and/or desirable. Therefore, the quality of the specification code and the formal techniques employed during transformations ("compilations") in order to deliver the hardware and software components of the system are receiving increasing focus in IoT chip application development.

2.2.2 *HLS Scheduling*

The HLS scheduling task belongs into two major categories: time-constrained scheduling and resource-constrained scheduling. Time-constrained scheduling aims to result into the lowest area or number of functional units, when the task is constrained by the max number of control steps (time constraint). Resource-constrained scheduling aims to produce the fastest schedule (the minimum number of control states) when the maximum number of hardware resources or hardware area is constrained (resource constraint). Integer linear programming (ILP) solutions have been proposed, however, their run time grows exponentially with the increase of design size, which makes them impractical. Heuristic methods have also been proposed to handle large designs and to provide sub-optimal but practical implementations. There are two heuristic scheduling approaches: constructive solutions

and iterative refinement. As-soon-as-possible (ASAP) and the as-late-as-possible (ALAP) scheduling both belong to the constructive approaches.

Operations that belong to the critical path of the design are not given any special priority over other operations in both ASAP and ALAP algorithms. Thus, excessive delay may result on the critical path, resulting into bad quality of the produced circuit implementation. LIST scheduling utilizes a global priority function to select the next operation to be scheduled. This global priority function can be either the mobility of the operation [1], or its urgency [2]. Force-directed scheduling [3] calculates the range of control steps for each operation between the operation's ASAP and ALAP state assignment. The algorithm then attempts to reduce the total number of functional units, so as to evenly distribute the operations of the same type into all of the available states of the range, producing better implementations against the rest of the heuristics, with an increased run time, however.

Constructive scheduling doesn't do any lookahead into future assignment of operations into the same control step, and this may lead to sub-optimal implementations. After an initial schedule is done by any of the above scheduling algorithms, then iteratively re-scheduling sequences of operations can maximally reduce the cost functions [4]. This is suitable for dataflow-oriented designs with linear control. For control-intensive designs, the use of loop pipelining [5] and loop folding [6] have been reported as scheduling tasks in the bibliography.

2.2.3 *Allocation and Binding Tasks*

Allocation determines the type of resource storage and functional units, selected from the library of components, to be assigned for each data object and operation of the input program. Allocation also calculates the number of resources of each type that are needed to implement every operation or data variable. Binding assigns operations, data variables, data structures, and data transfers onto functional units, storage elements (registers or memory blocks), and interconnections, respectively. Also binding makes sure that the design's functionality does not change by using the selected library components.

There are three categories of solutions to the allocation problem: constructive techniques, decomposition techniques, and iterative approaches. Constructive allocation techniques start with an empty implementation and progressively build the datapath and control parts of the implementation by adding more functional, storage, and interconnection elements while they traverse the CDFG or any other internal graph/representation format. Decomposition techniques divide the allocation problem into a sequence of well-defined independent sub-tasks. Each such sub-task is a graph-based theoretical problem which is solved with any of the three well-known graph methods: clique partitioning, the left-edge technique, and the weighted bipartite-matching technique. The task of finding the minimum cliques in the graph, which is the solution for the sub-tasks, is an NP-hard problem, so heuristic approaches [7] are utilized for allocation.

Because the conventional sub-task of storage allocation ignores the side-effects between the storage and interconnections allocation, when using the clique partitioning technique, graph edges are enhanced with weights that represent the effect on interconnection complexity. The left-edge algorithm is applied on the storage allocation problem, and it allocates the minimum number of registers [8]. A weighted, bipartite-matching algorithm is used to solve both the storage and functional unit allocation problems. First a bipartite graph is generated which contains two disjoint sets, e.g., one for variables and one for registers, or one for operations and one for functional units. An edge between one node of the one of the sets and one node of the other represents an allocation of, e.g., a variable to a register. The bipartite-matching algorithm considers the effect of register allocation on the design's interconnection elements, since the edges of the two sets of the graph are weighted [9]. The generated datapaths are improved iteratively, a simple assignment exchange, but using the pairwise exchange of the simulated annealing, or by using a branch-and-bound approach. The latter reallocates groups of elements of different types [10].

2.2.4 History of High-Level Synthesis Tools

HLS has been an active research field for about three decades. Early approaches of experimental synthesis tools that synthesized small subsets of programming constructs or proprietary modeling formats have emerged since the late 1980s. As an example, an early tool that generated hardware structures from algorithmic code, written in the PASCAL-like, digital system specification language (DSL) is reported in [11]. This synthesis tool performs the circuit compilation in two steps: first step is datapath synthesis which is followed by control synthesis. Examples of other behavioral circuit specification languages of that time, apart from DSL, were DAISY [12], ISPS [13], and MIMOLA [14].

In [15] the circuit to be synthesized is described with a combination of algorithmic and structural level code and then the PARSIFAL tool synthesizes the code into a bit-serial DSP circuit implementation. The PARSIFAL tool is part of a larger E-CAD system called FACE and which included the FACE design representation and design manager core. FACE and PARSIFAL were suitable for DSP pipelined implementations, rather than for a more general behavioral hardware models with hierarchy and complex control.

According to [16] scheduling first determines the propagation delay of each operation and then it assigns all operations into control steps (states) of a finite state machine. List scheduling uses a local priority function to postpone the assignment of operations into states, when resource constraints are violated. On the contrary, force-directed scheduling (FDS) tries to satisfy a global execution deadline (time constraint) as it minimizes the utilized hardware resources (functional units, registers, and busses). The force-directed list scheduling (FDLS) algorithm attempts to implement the fastest schedule while satisfying fixed hardware resource constraints.

The main HLS tasks in [17] include allocation, scheduling, and binding. According to [18] scheduling is finding the sequence of operations to execute in a specific order so as to produce a schedule of control steps with allocated operations in each step of the schedule; allocation defines the required number of functional, storage, and interconnect units; binding assigns operations to functional units, variables, and values to storage elements and the interconnections amongst them to form a complete working circuit that executes the functionality of the source behavioral model.

The V compiler [19] translates sequential descriptions into RTL models using parsing, scheduling, and resource allocation. The source sequential descriptions are written in the V language which includes queues, asynchronous calls, and cycle blocks and it is tuned to a kind of parallel hardware RTL implementations. The V compiler utilizes percolation scheduling [20] in order to achieve the required degree of parallelism by meeting time constraints.

A timing network is generated from the behavioral design in [21] and is annotated with parameters for every different scheduling approach. The scheduling approach in this work attempts to satisfy a given design cycle for a given set of resource constraints, using the timing model parameters. This approach uses an integer linear program (ILP) which minimizes a weighted sum of area and execution time of the implementation. According to the authors, their Symphony tool delivers better area and speed than ADPS [22]. This synthesis technique is suitable for dataflow designs (e.g., DSP blocks) and not for more general complex control flow designs.

The CALLAS synthesis framework [23] transforms algorithmic, behavioral VHDL models into VHDL RTL and gate netlists, under timing constraints. The generated circuit is implemented using a Moore-type finite state machine (FSM), which is consistent with the semantics of the VHDL subset used for the specification code. Formal verification techniques such as equivalence checking, which checks the equivalence between the original VHDL FSM and the synthesized FSM are used in the CALLAS framework by using the symbolic verifier of the circuit verification environment (CVE) system [24].

A methodological approach of designing and developing mixed hardware and software parts of a system known as hardware–software co-design has emerged about the same time as early HLS tools in the 1990s. The most known examples of this technology are reported in the following approaches.

The Ptolemy framework [25] allows for an integrated hardware–software co-design methodology from the specification through to synthesis of hardware and software components, simulation, and evaluation of the implementation. The tools of Ptolemy can synthesize assembly code for a programmable DSP core (e.g., DSP processor), which is built for a synthesis-oriented application. In Ptolemy, an initial model of the entire system is partitioned into the software and hardware parts which are synthesized in combination with their interface synthesis.

The COSYMA hardware–software co-synthesis framework [26] realizes an iterative partitioning process, based on hardware extraction algorithm which is driven by a cost function. The primary target in this work is to minimize customized hardware within microcontrollers but the same time to allow for space exploration

of large designs. The specialized co-processors of the embedded system can be synthesized using HLS tools. The specification language is based on C with various extensions. The generated hardware descriptions are in turn ported to the Olympus HLS tool [27]. The presented work included tests and experimental results based on a configuration of an embedded system, which is built around the Sparc microprocessor.

Co-synthesis and hardware–software partitioning are executed in combination with control parallelism transformations in [28]. The hardware–software partition is defined by a set of application-level functions which are implemented with application-specific hardware. The control parallelism is defined by the interaction of the processes of the functional behavior of the specified system. The system behavior is modeled using a set of communicating sequential processes [29]. Each process is then assigned either to hardware or to software implementation.

A hardware–software co-design methodology, which employs synthesis of heterogeneous systems, is presented in [30]. The synthesis process is driven by timing constraints which drive the mapping of tasks onto hardware or software parts so that the performance requirements of the intended system are met. This method is based on using modeling and synthesis of programs written in the HardwareC language. An example application which was used to test the methodology in this work was an Ethernet-based network co-processor.

2.2.5 Next Generation High-Level Synthesis Tools

More advanced methodologies appeared at the late 1990s and they featured enhanced input code sets as well as improved scheduling and other optimization algorithms. The CoWare hardware–software co-design environment [31] is based on a data model that allows the user to specify, simulate, and produce heterogeneous implementations from heterogeneous specification source models. This design approach develops telecommunication systems that contain DSP, control loops, and user interfaces. The synchronous dataflow (SDF) type of algorithms found in a category of DSP applications are synthesized into hardware from languages such as SILAGE [32], DFL [33], and LUSTRE [34]. In contrast to this, DDF algorithms consume and produce tokens that are data-dependent, and thus they allow for complex if-then-else and while loop control constructs. CAD systems that allow for specifying both SDF and DDF algorithms and run scheduling are the DSPstation from Mentor Graphics [35], PTOLEMY [36], GRAPE-II [37], COSSAP from Synopsys, and SPW from the Alta group [38].

C programs that include dynamic memory allocation, pointers, and the functions malloc and free are mapped onto hardware in [39]. The SpC tool which was developed in this work resolves pointer variables at compile time and thus C functional models are synthesized into Verilog hardware models. The synthesis of functions in C, and therefore the resolution of pointers and malloc/free inside of functions, is, however, not included yet in this work, but left as future work.

The different techniques and optimizations described above have been implemented using the SUIF compiler environment [40].

A heuristic for scheduling behavioral specifications that include a lot of conditional control flow is presented in [41]. This heuristic is based on a powerful intermediate design representation called hierarchical conditional dependency graph (HCDG). HCDG allows chaining and multicycling, and it enables advanced techniques such as conditional resource sharing and speculative execution, which are suitable for scheduling conditional behaviors. The HLS techniques in this work were implemented in a prototype graphical interactive tool called CODESIS which used HCDG as its internal design representation. The tool generates VHDL or C code from the HCDG, but no translation of standard programming language code into HCDG is known so far.

A coordinated set of coarse-grain and fine-grain parallelizing HLS transformations on the input design model are discussed in [42]. These transformations are executed in order to deliver synthesis results that don't suffer from the negative effects of complex control constructs in the specification code. All of the HLS techniques in this work were implemented in the SPARK HLS tool, which transforms specifications in a small subset of C into RTL VHDL hardware models. SPARK utilizes both control/dataflow graphs (CDFGs) and an encapsulation of basic design blocks inside hierarchical task graphs (HTGs), which enable coarse-grain code restructuring such as loop transformations and an efficient way to move operations across large pieces of code. Nevertheless, SPARK is not designed to process conditional code where the iteration limits are not known at compile time, such as while loops.

Typical HLS tasks such as scheduling, resource allocation, module binding, module selection, register binding, and clock selection are executed simultaneously in [43] so as to achieve better optimization in design energy, power, and area. The scheduling algorithm utilized in this HLS methodology applies concurrent loop optimization and multicycling and it is driven by resource constraints. The state transition graph (STG) of the design is simulated in order to generate switched capacitance matrices. These matrices are then used to estimate power/energy consumption of the design's datapath. Nevertheless, the input to the HLS tool is not programming language code but a complex proprietary format representing an enhanced CDFG as well as an RTL design library and resource constraints.

An incremental floorplanner is described in [44] which combines an incremental behavioral and physical optimization into HLS. These techniques were integrated into an existing interconnect-aware HLS tool called ISCALP [45]. The new combination was named IFP-HLS (incremental floorplanner high-level synthesis) tool, and it attempts to concurrently improve the design's schedule, resource binding, and floorplan, by integrating high-level and physical design algorithms.

Huang et al. [46] discuss a HLS methodology which is suitable for the design of distributed logic and memory architectures. Beginning with a behavioral description of the system in C, the methodology starts with behavioral profiling in order to extract simulation statistics of computations and references of array data. Then array data are distributed into different partitions. An industrial tool called Cyber [47]

was developed which generates a distributed logic/memory micro-architecture RTL model, which is synthesizable with existing RTL synthesizers, and which consists of two or more partitions, depending on the clustering of operations that was applied earlier.

A system specification containing communicating processes is synthesized in [48]. The impact of the operation scheduling is considered globally in the system critical path (as opposed to the individual process critical path), in this work. It is argued by the authors in this work that this methodology allocates the resources where they are mostly needed in the system, which is in the critical paths, and in this way it improves the overall multi-process designed system performance.

The work in [49] contributes towards incorporating memory access management within a HLS design flow. It mainly targets DSP applications but also other streaming applications can be included along with specific performance constraints. The synthesis process is performed on the extended dataflow graph (EDFG) which is based on the signal flow graph. Mutually exclusive scheduling methods [50, 51] are implemented with the EDFG. The graph which is processed by a number of annotations and improvements is then given to the GAUT HLS tool [52] to perform operator selection and allocation, scheduling and binding.

A combined execution of operation decomposition and pattern-matching techniques is targeted to reduce the total circuit area in [53]. The datapath area is reduced by decomposing multicycle operations, so that they are executed on monocycle functional units (FUs that take one clock cycle to execute and deliver their results). A simple formal model that relies on an FSM-based formalism for describing and synthesizing on-chip communication protocols and protocol converters between different bus-based protocols is discussed in [54]. The utilized FSM-based format is at an abstraction level which is low enough so that it can be automatically translated into HDL implementations. The generated HDL models are synthesizable with commercial tools. Synchronous FSMs with bounded counters that communicate via channels are used to model communication protocols. The model devised in this work is validated with an example of communication protocol pairs which included AMBA APB and ASB. These protocols are checked regarding their compatibility, by using the formal model.

The methodology of SystemCoDesigner [55] uses an actor-oriented approach so as to integrate HLS into electronic system level (ESL) design space exploration tools. The design starts with an executable SystemC system model. Then, commercial synthesizers such as Forte's Cynthesizer are used in order to generate hardware implementations of actors from the behavioral model. This aids the design space exploration in finding the best candidate architectures (mixtures of hardware and software modules). After deciding on the chosen solution, the suitable target platform is then synthesized with the implementations of the hardware and software parts. The final step of this methodology is to generate the FPGA-based SoC implementation from the chosen hardware/software solution. Based on the proposed methodology, it seems that SystemCoDesigner method is suitable for stream-based applications, found in areas such as DSP, image filtering, and communications.

A formal approach is followed in [56] which is used to prove that every HLS translation of a source code model produces an RTL model that is functionally equivalent to the one in the behavioral input to the HLS tools. This technique is called translation validation and it has been maturing via its use in the optimizing software compilers. The validating system in this work is called SURYA, it is using the Symplify theorem prover and it was used to validate the SPARK HLS tool. This validation experiment with Symplify discovered two bugs in the SPARK compilations.

The replacement of flip-flop registers with latches is proposed in [57] in order to yield better timing in the implemented designs. The justification for this is that latches are inherently more tolerant to process variations than flip-flops. The related design techniques were integrated into a tool called HLS-1. HLS-1 translates behavioral VHDL code into a synthesized netlist. Nevertheless, handling a design where registers are implemented with latches instead of edge-triggered flip-flops is generally considered to be cumbersome due to the complicated timing behavior of latches.

2.3 Synthesis for Low Power

Many portable and embedded computing systems and applications such as mobile (smart) phones, PDAs, etc., require design for low power and therefore synthesis for low energy is becoming very important in the whole area of VLSI and embedded system design. In any case, and particularly for IoT devices the majority of custom functions that are implemented in special-purpose hardware imply that they consume much less power than microcontroller equivalent programs. Therefore it is mandatory that most of the special functions in IoT devices found in areas of data compression, security, media playing, image/audio (de-)coding must be implemented using advanced synthesis techniques into specialized hardware for low power consumption and increased security gains.

During the last decade industry and academia invested on significant amounts of research regarding VLSI techniques and HLS for low power design. In order to achieve low energy in the results of HLS and system design, new techniques that help to estimate power consumption at the high-level description level are needed, and they will be a great aid in delivering systems with reduced power consumption such as IoT devices running on batteries. In [58], switching activity and power consumption are estimated at the RTL level taking also into account the glitching activity on a number of signals of the datapath and the controller. The spatial locality, the regularity, the operation count, and the ratio of critical path to available time are identified in [59] with the aim to reduce the power consumption of the interconnections. The HLS scheduling, allocation, and binding tasks consider such algorithmic statistics and properties in order to reduce the fanins and fanouts of the interconnect wires. This will result into reducing the complexity and the power consumed on the capacitance of the interconnection buses [60].

The effect of the controller on the power consumption of the datapath is considered in [61]. Pipelining and module selection was proposed in [62] for low power consumption. The activity of the functional units was reduced in [63] by minimizing the transitions of the functional unit's inputs. This was utilized in a scheduling and resource binding algorithm, in order to reduce power consumption. In [64] the DFG is simulated with profiling stimuli, provided by the user, in order to measure the activity of operations and data carriers. Then, the switching activity is reduced by selecting a special module set and schedule. Reducing supply voltage, disabling the clock of idle elements, and architectural tradeoffs were utilized in [65] in order to minimize power consumption within HLS.

The energy consumption of memory subsystem and the communication lines within a multiprocessor system-on-a-chip (MPSoC) is addressed in [66]. This work targets streaming applications such as image and video processing that have regular memory access patterns. The way to realize optimal solutions for MPSoCs is to execute the memory architecture definition and the connectivity synthesis in the same step.

The above research approaches all attempt to target energy efficiency at the device consumption level and improve architectural and device level techniques for reducing it. It is absolutely necessary to transition from traditional RTL design techniques into HLS-based development so that to automate the design and deliver higher quality of implementations that are suitable for IoT embedded and battery-running devices. Also, by using formal and integrated design and verification techniques a great deal of project time is saved so that necessary focus can be shifted from repetitive verification cycles into advanced (micro-)architectural issues of the system, for improved performance and power consumption.

2.4 The C-Cubed Hardware Synthesis Flow

So far in this chapter related work in HLS methodologies for embedded IoT systems was reviewed. From this section onwards, a particular, formal HLS methodology is analyzed and explained, which is directly applicable on embedded system design, and it has been developed by the author of this chapter. The techniques of this work include the front-end compilers which are based on formal compiler generators, the Formal Intermediate Format (FIF) which encapsulates in a formal way the attributes of the input algorithms, and the back-end compiler which is built with formal logic programming relations (or facts in the Prolog language terminology).

The FIF¹ was invented and designed by the author of this chapter as a tool and media for the design encapsulation and the HLS transformations in the C-Cubed

¹The Formal Intermediate Format is patented with patent number: 1006354, 15/4/2009, from the Greek Industrial Property Organization.

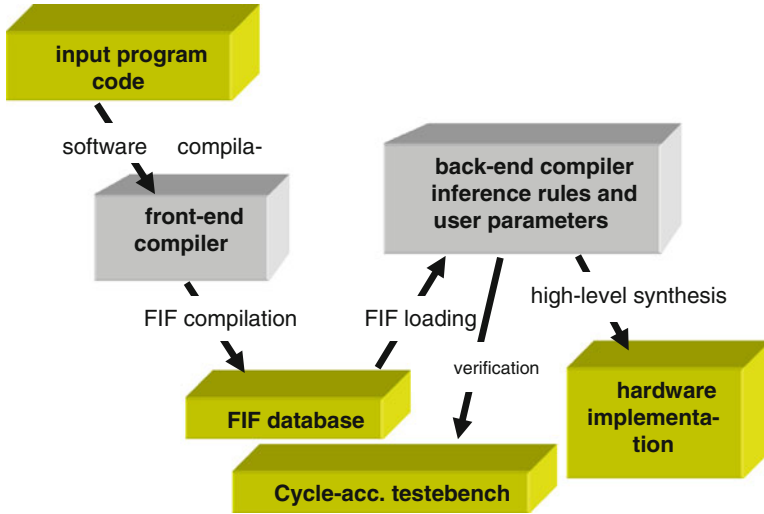


Fig. 2.1 The C-Cubed hardware synthesis flow and tools

(Custom Co-processor Compilation) hardware compilation tool.² A near-complete analysis of FIF syntax and semantics can be found in [67]. The formal methodology discussed here is based on using predicate logic to describe the intermediate representations and transformations of the compilation steps, and the resolution of a set of transformation Horn clauses [68] is used, as the building blocks of the prototype hardware compiler.

The front-end compiler translates the input program code into the FIF's logic statements (logic facts). The inference logic rules of the back-end compiler transform the FIF facts into the hardware implementations. There is one-to-one correspondence between the source specification's subroutines and the generated hardware modules. The source code subroutines can be hierarchical, and this hierarchy is maintained in the generated hardware implementation. Each generated hardware model is an FSM-controlled custom processor (or co-processor, or accelerator), that executes a specific task, described in the source program code. This hardware synthesis flow is depicted in Fig. 2.1. The generated hardware modules specify a standalone function (e.g., DSP filter) and they are coded in the VHDL or the Verilog HDL languages. For verification purposes, and for every hardware module, a fast cycle-accurate testbench, coded in ANSI-C, is generated from the same internal formal model of the FSM and the datapath of the co-processor.

Essentially the front-end compilation resembles software program compilation and the back-end compilation executes formal transformation tasks that are

²This hardware compiler method is patented with patent number: 1005308, 5/10/2006, from the Greek Industrial Property Organization.

normally found in HLS tools. This whole compilation flow is a formal transformation process, which converts the source code programs into implementable RTL VHDL hardware accelerator models. If there are function calls in the specification code, then each subprogram call is transformed into an interface event in the generated hardware FSM. The interface event is used so that the “calling” accelerator uses the “services” of the “called” accelerator, as it is depicted in the source code hierarchy as well.

2.5 Back-End Compiler Inference Logic Rules

The back-end compiler consists of a very large number of logic rules. These logic rules are coded with logic programming techniques, which are used to implement the HLS transformations and other processes of the back-end compilation phase. As an example, one of the latter processes reads and incorporates the FIF tables’ facts into the compiler’s internal inference engine of logic predicates and rules [68]. The back-end compiler rules are given as a great number of definite clauses of the following equation:

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_n \text{ (where } n \geq 0) \quad (2.1)$$

where \leftarrow is the logical implication symbol ($A \leftarrow B$ means that if B applies then A applies), and A_0, \dots, A_n are atomic formulas (logic facts) of the equation:

$$\text{predicate_symbol}(\text{Var_1}, \text{Var_2}, \dots, \text{Var_N}) \quad (2.2)$$

where the positional parameters $\text{Var_1}, \dots, \text{Var_N}$ of the above predicate “predicate_symbol” are either variable names (in the case of the back-end compiler inference rules), or constants (in the case of the FIF table statements). The predicate syntax in Eq. (2.2) is typical of the way of the FIF facts and other facts interact with each other, they are organized and they are used internally in the inference engine. Thus, the hardware descriptions are generated as “conclusions” of the inference engine upon the FIF “facts.” This is done in a formal way from the input programs by the back-end phase, which turns the overall transformation into a provably correct compilation process. In essence, the FIF file consists of a number of such atomic formulas, which are grouped in the FIF tables. Each such table contains a list of homogeneous facts which describe a certain aspect of the compiled program. For example, all prog_stmt facts for a given subprogram are grouped together in the listing of the program statements table.

In the back-end compiler “conclusions” the correct-by-construction hardware implementations and the cycle-accurate testbench are included, as custom hardware micro-architectures. This along with other features makes the C-Cubed tool very suitable for designing custom hardware blocks of IoT embedded devices and peripherals. Experiments with the tool have shown that it is very suitable for

small and big data coding, secure functions and cryptography, DSP and computer graphics, as well as other mathematical blocks, all of which are used in today's embedded systems.

2.6 Inference Logic and Back-End Transformations

The inference engine of the back-end compiler consists of a great number of logic rules (like the one in Eq. (2.1)) which conclude on a number of input logic predicate facts and produce another set of logic facts and so on. Eventually, the inference logic rules produce the logic predicates that encapsulate the writing of RTL VHDL hardware co-processor models. These hardware models are directly implementable to any hardware (e.g., ASIC or FPGA) technology, since they are technology and platform-independent. For example, generated RTL models produced in this way from the prototype compiler were synthesized successfully into hardware implementations using the Synopsys DC Ultra, the Xilinx ISE, and the Mentor Graphics Precision software without the need of any manual alterations of the produced RTL VHDL code. In the following Eq. (2.3) an example of such an inference rule is shown:

$$\begin{aligned} \text{dont_schedule}(\text{Operation1}, \text{Operation2}) \leftarrow \\ \text{examine}(\text{Operation1}, \text{Operation2}), \\ \text{predecessor}(\text{Operation1}, \text{Operation2}). \end{aligned} \quad (2.3)$$

The meaning of this rule that combines two input logic predicate facts to produce another logic relation (`dont_schedule`) is that when two operations (`Operation1` and `Operation2`) are examined and the first is a predecessor of the second (in terms of data and control dependencies), then don't schedule them in the same control step. This rule is part of a parallelizing optimizer which is called "PARCS" (meaning: Parallel, Abstract Resource—Constrained Scheduler).

The way that the inference engine rules (predicates relations—productions) work is depicted in Fig. 2.2. The last produced (from its rule) predicate fact is the VHDL RTL writing predicate at the top of the diagram. Right below level 0 of predicate production rule there is a rule at the -1 level, then level -2 , and so on. The first predicates that are fed into this engine of production rules belong to level $-K$, as shown in this figure. Level $-K$ predicate facts include of course the FIF facts that are loaded into the inference engine along with the other predicates of this level.

In this way, the back-end compiler works with inference logic on the basis of predicate relation rules and, therefore, this process is a formal transformation of the FIF source program definitions into the hardware accelerator (implementable) models. Of course in the case of the prototype compiler, there is a very large number of predicates and their relation rules that are defined inside the implementation code of the back-end compiler, but the whole concept of implementing this phase is as shown in Fig. 2.2. The user of the back-end compiler can select certain environment

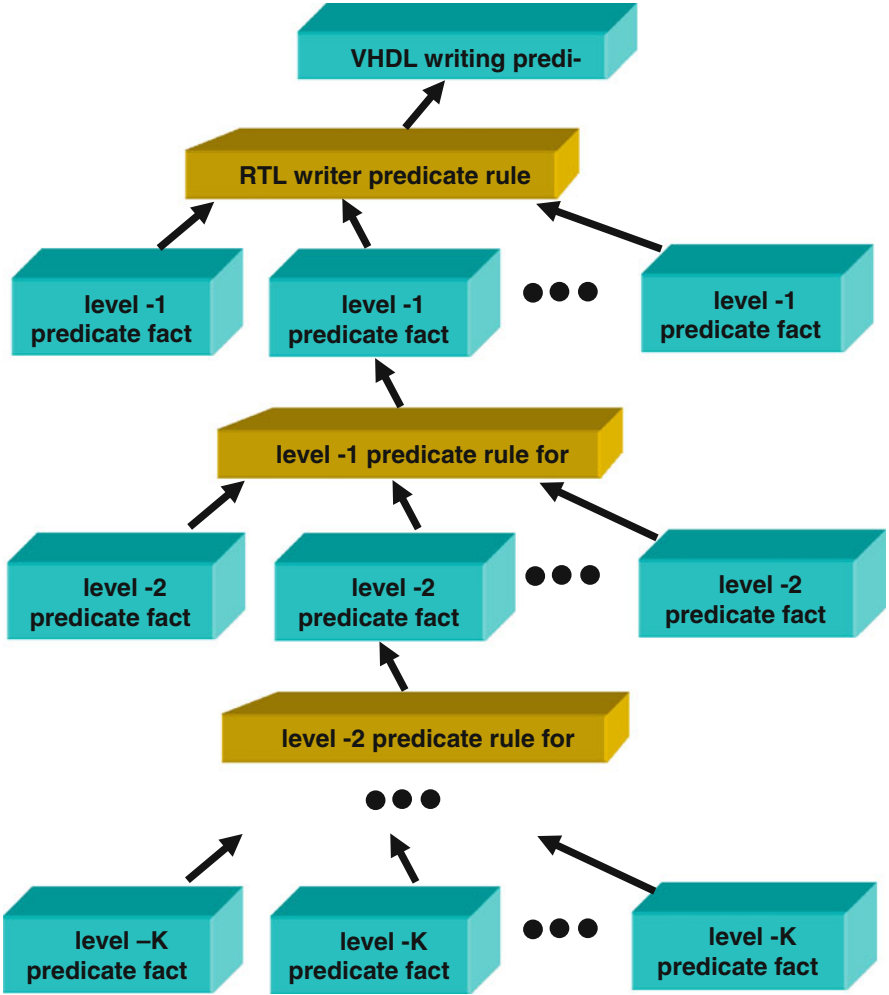


Fig. 2.2 The back-end inference logic rules structure

command list options as well as build an external memory port parameter file as well as drive the compiler’s optimizer with specific resource constraints of the available hardware operators.

The most important of the back-end compilation stages can be seen in Fig. 2.3. The compilation process starts with the loading of the FIF facts into the inference rule engine. After the FIF database is analyzed, the local data object, operation, and initial state lists are built. Then the environment options are read and the temporary lists are updated with the special (communication) operations as well as the predecessor and successor dependency relation lists. After the complete initial schedule is built and concluded, the PARCS optimizer is run on it, and the optimized

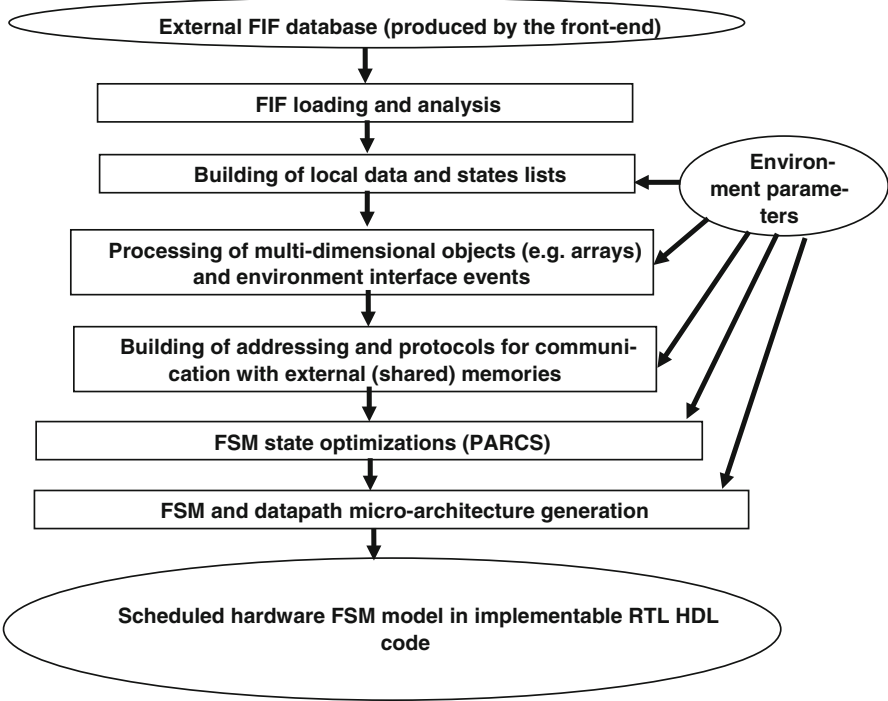


Fig. 2.3 The processing stages of the back-end compiler

schedule is delivered to the micro-architecture generator. The transformation is concluded with the formation of the FSM and datapath implementation and the writing of the RTL VHDL model for each accelerator that is defined in each subprogram of the source code program.

A separate hardware accelerator model is generated from each subprogram in the system model code. All of the generated hardware models are directly implementable into hardware using commercial CAD tools, such as the Synopsys DC-ultra, the Xilinx ISE, and the Mentor Graphics Precision RTL synthesizers. Also the hierarchy of the source program modules (subprograms) is maintained and the generated accelerators may be hierarchical. This means that an accelerator can invoke the services of another accelerator from within its processing states, and that other accelerator may use the services of yet another accelerator, and so on. In this way, a subprogram call in the source code is translated into an external co-processor interface event of the corresponding hardware accelerator.

2.7 The PARCS Optimizer

The PARCS scheduler aggressively attempts to schedule as many as possible operations in the same control step. The only limits to this are the data and control dependencies as well as the optional resource (operator) constraints, which are provided by the user.

The pseudo-code for the main procedures of the PARCS scheduler is shown in Fig. 2.4. All of the predicate rules (like the one in Eq. (2.1)) of PARCS are part of the inference engine of the back-end compiler. A new design to be synthesized is loaded via its FIF into the back-end compiler's inference engine. Hence, the FIF's facts as well as the newly created predicate facts from the so far logic processing "drive" the logic rules of the back-end compiler which generate provably correct hardware architectures. It is worthy to note that although the HLS transformations are implemented with logic predicate rules, the PARCS optimizer is very efficient and fast. In most of benchmark cases that were run through the prototype hardware compiler flow, compilation did not exceed 1–10 min of run time and the results of the compilation were very efficient as explained below.

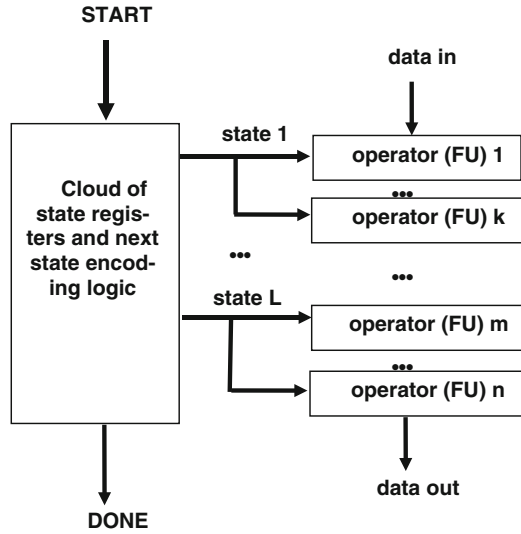
2.8 Generated Hardware Architectures

The back-end stage of micro-architecture generation can be driven by command-line options. One of the options, e.g., is to generate massively parallel architectures. The results of this option are shown in Fig. 2.5. This option generates a single process—FSM VHDL or Verilog description with all the data operations being dependent on

1. start with the initial schedule (including the special external port operations)
2. Current PARCS state \leftarrow 1
3. Get the 1st state and make it the current state
4. Get the next state
5. Examine the next state's operations to find out if there are any dependencies with the current state
6. If there are no dependencies then absorb the next state's operations into the current PARCS state; If there are dependencies then finalize the so far absorbed operations into the current PARCS state, store the current PARCS state, PARCS state \leftarrow PARCS state + 1; make next state the current state; store the new state's operations into the current PARCS state
7. If next state is of conditional type (it is enabled by guarding conditions) then call the conditional (true/false branch) processing predicates, else continue
8. If there are more states to process then go to step 4, otherwise finalize the so far operations of the current PARCS state and terminate

Fig. 2.4 Pseudo-code of the PARCS scheduling algorithm

Fig. 2.5 Massively parallel micro-architecture generation option



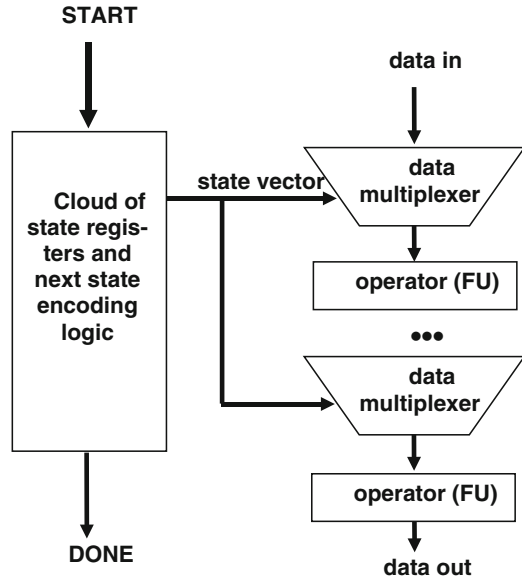
different machine states. This implies that every operator is enabled by single wire activation commands that are driven by different state register values. This in turn means that there is a redundancy in the generated hardware, in a way that during part of execution time, a number of state-dedicated operators remain idle. However, this redundancy is balanced by the fact that this option achieves the fastest clock cycle, since the state command encoder, as well as the data multiplexers are replaced by single wire commands which don't exhibit any additional delay, and this option is very suitable to implement on large ASICs with plenty of resources.

Another micro-architecture option is the generation of traditional FSM + datapath-based VHDL/Verilog models. The results of this option are shown in Fig. 2.6. With this option activated the generated VHDL/Verilog models of the hardware accelerators include a next state process as well as signal assignments with multiplexing which correspond to the input data multiplexers of the activated operators. Although this option produces smaller hardware structures (than the massively parallel option), it can exceed the target clock period due to larger delays through the data multiplexers that are used in the datapath of the accelerator.

Using the above micro-architecture options, the user of the CCC HLS tool can select various solutions between the fastest and larger massively parallel micro-architecture, which may be suitable for richer technologies in terms of operators such as large ASICs, and smaller and more economic (in terms of available resources) technologies such as smaller FPGAs.

As it can be seen in Figs. 2.5 and 2.6, the produced co-processors (accelerators) are initiated with the input command signal START. Upon receiving this command the co-processors respond to the controlling environment using the handshake output signal BUSY and right after this, they start processing the input data in order to produce the results. This process may take a number of clock cycles

Fig. 2.6 The traditional FSM + datapath generated micro-architecture option



and it is controlled by a set of states (discrete control steps). When the co-processors complete their processing, they notify their environment with the output signal DONE. In order to conclude the handshake the controlling environment (e.g., a controlling central processing unit) responds with the handshake input RESULTS_READ, to notify the accelerator that the processed result data have been read by the environment. This handshake protocol is also followed when one (higher-level) co-processor calls the services of another (lower-level) co-processor. The handshake is implemented between any number of accelerators (in pairs) using the START/BUSY and DONE/RESULTS_READ signals. Therefore, the set of executing co-processors can also be hierarchical in this way.

Other environment options, passed to the back-end compiler, control the way that the data object resources are used, such as registers and memories. Using a memory port configuration file, the user can determine that certain multi-dimensional data objects, such as arrays and array aggregates, are implemented in external (e.g., central, shared) memories (e.g., system RAM). Otherwise, the default option remains that all data objects are allocated to hardware (e.g., on-chip) registers. All of the related memory communication protocols and hardware ports/signals are automatically generated by the back-end synthesizer, and without the need for any manual editing of the RTL code by the user. Both synchronous and asynchronous memory communication protocol generation are supported.

2.9 Generated Hardware Execution Platform

The generated hardware modules can be placed inside the computing environment that they accelerate or can be executed standalone. For every subprogram in the source specification code one co-processor is generated to speed up (accelerate) or just execute the particular system task. The whole system (both hardware and software models) is modeled in algorithmic ADA or C code which can be compiled and executed with the host compiler and linker to run and verify the operation of the whole system at the program code level. In this way, extremely fast verification can be achieved at the algorithmic level. It is evident that such behavioral (high-level) compilation and execution is orders of magnitude faster than conventional RTL simulations. Moreover, now C-Cubed automatically generates cycle-accurate C testbenches from exactly the same FSM information that generates the HDL code. Therefore, using compile and execute these testbenches can be executed on the host computer in a rapid and correct manner.

After the required co-processors are specified, coded in ADA, generated with the prototype hardware compiler, and implemented with commercial back-end tools, they can be downloaded into the target computing system (if the target system includes FPGAs) and executed to accelerate certain system tasks. This process is shown in Fig. 2.7. The accelerators can communicate with each other and with the host computing environment using synchronous handshake signals and connections with the system's handshake logic.

2.10 Experimental Results and Conclusions

There have been a great deal of design and verification experiments with the C-Cubed framework. In all the experiments it was found that the quality of the generated HDL modules is very high with increased readability of the code. Among the many experiments, three small benchmarks are analyzed in this paragraph: a computer graphics algorithm, a DSP FIR (finite impulse response) filter, and the classical high-level synthesis benchmark, the second order differential equation approximation solver. Moreover, the statistics of two more benchmarks, an RSA crypto-processor and a complex MPEG engine, are included in the discussion of this chapter.

Table 2.1 shows state reduction statistics with the C-Cubed's optimizer, PARCS. Impressive state reduction is achieved, up to 41 %, even with a complex conditional structure which is found in the line-drawing design (computer graphics benchmark). Due to the formal nature of the C-Cubed synthesizer only high-level source code—level compile and execute verifications are needed. However, in order to prove our argument in practice all synthesized RTL modules were simulated and their behavior matched, as expected, the behavior of the source code programs. Figure 2.8 shows a snapshot of the RTL simulation of the computer graphics hardware module, near the time where the line's pixels are written into the external memory.

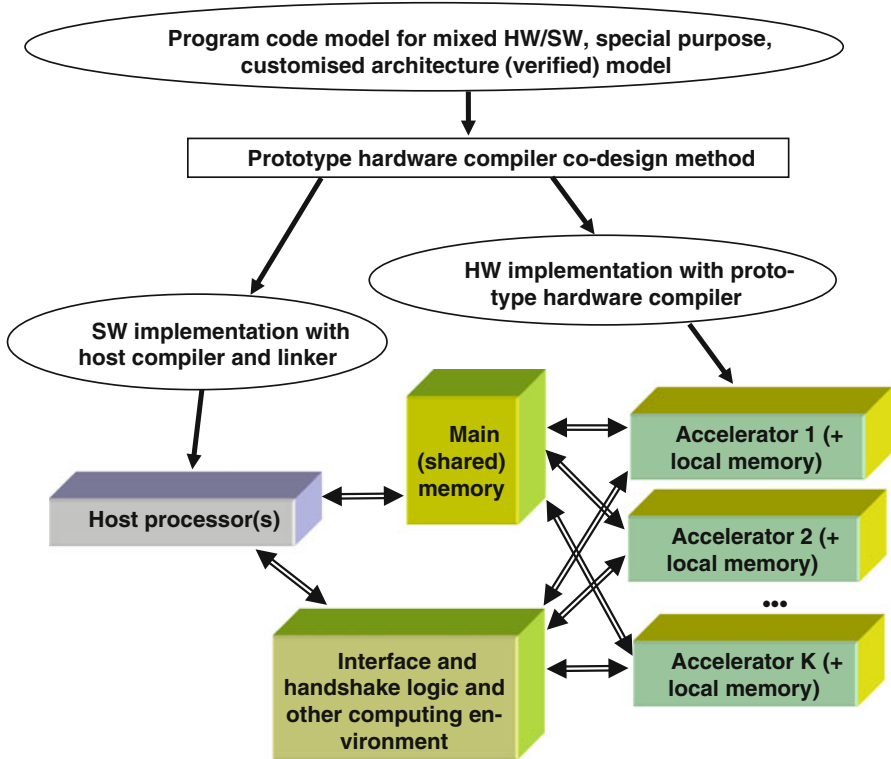


Fig. 2.7 Host computing environment and hardware execution configuration

Table 2.1 FSM state statistics before and after optimization with PARCS

Module name	Initial schedule states	PARCS states	State reduction percentage (%)
DSP FIR filter processor	17	10	41
RSA crypto-processor	16	11	31
Computer graphics design	17	10	41
MPEG top routine (with external memory)	462	343	26
Differential equation solver	20	13	35

The verification flow in the C-Cubed framework is formal and integrated with the synthesis flow. The fact that the input to the tools is executable allows us to compile the source ADA or C and co-simulate with the testbench. Also, we can include commands in the high-level testbench in order to automatically and formally compare simulation outputs of the high-level code with that of the RTL code. The structure and flow of this cross-checking verification is shown in Fig. 2.9.

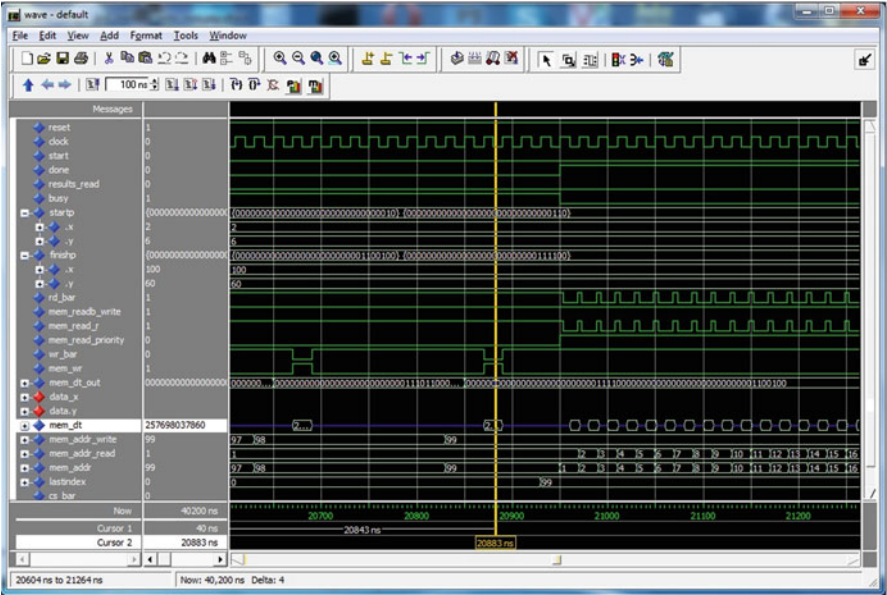


Fig. 2.8 RTL simulation snapshot of the line-drawing benchmark output

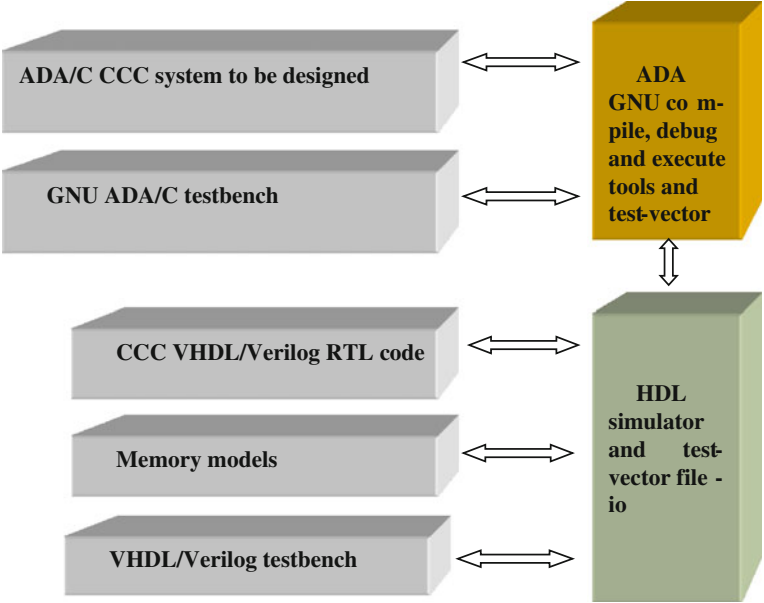


Fig. 2.9 ADA to VHDL cross-verification flow

Apart from the high-level behavioral testbench (at the source code level) the C-Cubed framework recently offered a formal verification option, by compiling and rapidly executing the FSM by means of a cycle-accurate testbench (FSM model) in the C language which is automatically generated by the back-end compiler using the same internal intelligent FSM models of the optimized hardware modules. The cycle-accurate testbench allows for setting up inputs, reading outputs and registers, resetting the engine, and moving to the next state by pressing corresponding buttons on the keyboard. Thus it is very easy to use and it can increase the confidence of the provably correctness of the generated FSM (co-processor or standalone custom logic).

As an example Fig. 2.10 shown the execution screen of the high-level testbench of the differential equation solver benchmark.

Figure 2.11 shows the RTL simulation of the same benchmark (the result is obviously the same as with the high-level testbench).

The start of the cycle-accurate testbench simulation of this benchmark is shown in Fig. 2.12, where the circuit’s inputs are set. After going through the FSM’s states the screen in Fig. 2.13 shows reading of the outputs which give the same results as the other verification runs for this test (as it was expected).

The same flow was confirmed for all of this work’s benchmarks, but due to limitations in paper length they are omitted.

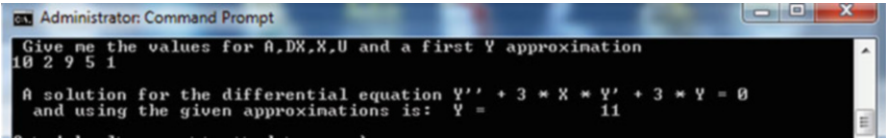


Fig. 2.10 High-level testbench execution of the differential equation solver

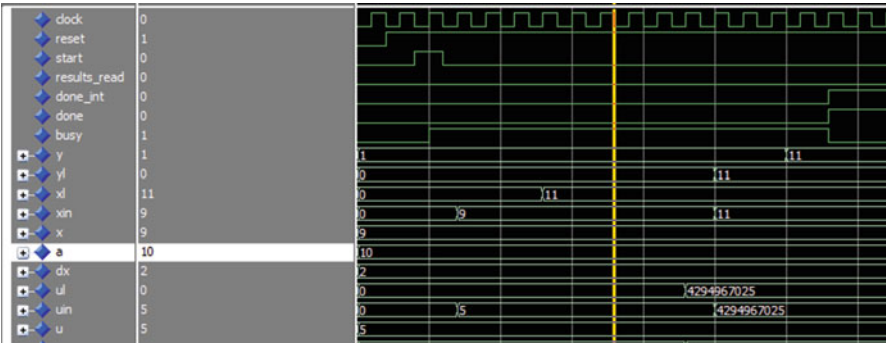


Fig. 2.11 RTL simulation of the differential equation solver


```

Administrator: Command Prompt
> type a simulator option: r(eset), n(ext state), i(nput settings) o(utput values)
> or q(uit): i
executing simulator body
reseting local signals and variables
type a simulator option: r(eset), n(ext state), i(nput settings) o(utput values)
> or q(uit): i
executing simulator body
give me the value of input a10
a = 10
give me the value of input dx2
dx = 2
give me the value of input x9
x = 9
give me the value of input u5
u = 5
give me the value of input y1
y = 1

```

Fig. 2.12 Setting the inputs of the cycle-accurate diff.eq.solver benchmark

```

Administrator: Command Prompt
var1 is false so next state = 12
type a simulator option: r(eset), n(ext state), i(nput settings) o(utput values)
> or q(uit): n
executing simulator body
last state 12
writing the design's outputs and synchronizing with the outside world...
by pressing n you move to state 0
type a simulator option: r(eset), n(ext state), i(nput settings) o(utput values)
> or q(uit): o
executing simulator body
the values of outputs are :
the value of output y = 11

```

Fig. 2.13 Reading the outputs of the cycle-accurate diff.eq.solver benchmark

2.11 Conclusions and Future Work

The major contribution of this work is an integrated, formal, rapid, and automated methodology and set of tools for the automatic synthesis of custom hardware, which can be used in embedded IoT devices. All of the above characteristics of the presented method make it suitable for short project time and limited project budget for designing custom circuit blocks. It is expected that the IoT industry will be benefited the most by adopting and using such HLS methods, which will make it competitive to the hard international economy.

Due to the nature of the C-Cubed tools implementation they are particularly suitable for future extensions and experiments such as low power design, SystemC testbench generation, and other language input/output interfaces. Also, continuous improvements of the PARCS scheduler and the associated synthesis transformations are envisaged and they are planned for the future.

References

1. B. Pangrle, D. Gajski, Design tools for intelligent silicon compilation. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **6**(6), 1098–1112 (1987)

2. E. Girczyc, R. Buhr, J. Knight, Applicability of a subset of Ada as an algorithmic hardware description language for graph-based hardware compilation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **4**(2), 134–142 (1985)
3. P. Paulin, J. Knight, Algorithms for high-level synthesis. *IEEE Des. Test Comput.* **6**(6), 18–31 (1989)
4. I. Park, C. Kyung, Fast and near optimal scheduling in automatic data path synthesis, in *Proceedings of the Design Automation Conference (DAC)*, pp. 680–685, San Francisco, CA, 1991
5. N. Park, A. Parker, Sehwa: a software package for synthesis of pipelined data path from behavioral specification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **7**(3), 356–370 (1988)
6. E. Girczyc, Loop winding—a data flow approach to functional pipelining, in *Proceedings of the International Symposium on Circuits and Systems*, pp. 382–385, Philadelphia, PA, May 1987
7. C. Tseng, D. Siewiorek, Automatic synthesis of data path on digital systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **5**(3), 379–395 (1986)
8. F. Kurdahi, A. Parker, REAL: a program for register allocation, in *Proceedings of the Design Automation Conference (DAC)*, pp. 210–215, Miami Beach, FL, June 1987
9. C. Huang, Y. Chen, Y. Lin, Y. Hsu, Data path allocation based on bipartite weighted matching, in *Proceedings of the Design Automation Conference (DAC)*, pp. 499–504, Orlando, FL, June 1990
10. F. Tsay, Y. Hsu, Data path construction and refinement, in *Digest of Technical Papers, International Conference on Computer-Aided Design (ICCAD)*, pp. 308–311, Santa Clara, CA, November 1990
11. R. Camposano, W. Rosenstiel, Synthesizing circuits from behavioral descriptions. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **8**(2), 171–180 (1989)
12. S. Johnson, *Synthesis of Digital Designs from Recursion Equations* (MIT Press, Cambridge, MA, 1984)
13. M. Barbacci, G. Barnes, R. Cattell, D. Siewiorek, The ISPS computer description language. Report CMU-CS-79-137, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1979
14. P. Marwedel, The MIMOLA design system: tools for the design of digital processors, in *Proceedings of the 21st Design Automation Conference (DAC)*, pp. 587–593, IEEE Press, Piscataway, NJ, 1984
15. A. Casavant, M. D’Abreu, M. Dragomirecky, D. Duff, J. Jasica, M. Hartman, K. Hwang, W. Smith, A synthesis environment for designing DSP systems. *IEEE Des. Test Comput.* **6**(2), 35–44 (1989)
16. P. Paulin, J. Knight, Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **8**(6), 661–679 (1989)
17. D. Gajski, L. Ramachandran, Introduction to high-level synthesis. *IEEE Des. Test Comput.* **11**(4), 44–54 (1994)
18. R. Walker, S. Chaudhuri, Introduction to the scheduling problem. *IEEE Des. Test Comput.* **12**(2), 60–69 (1995)
19. V. Berstis, The V compiler: automatic hardware design. *IEEE Des. Test Comput.* **6**(2), 8–17 (1989)
20. J. Fisher, Trace Scheduling: a technique for global microcode compaction. *IEEE Trans. Comput.* **C-30**(7), 478–490 (1981)
21. A. Kuehlmann, R. Bergamaschi, Timing analysis in high-level synthesis, in *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design (ICCAD ‘92)*, pp. 349–354, Los Alamitos, CA, 1992
22. C. Papachristou, H. Konuk, A linear program driven scheduling and allocation method followed by an interconnect optimization algorithm, in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC)*, Orlando, Florida, USA, pp. 77–83, June 1990

23. J. Biesenack, M. Koster, A. Langmaier, S. Ledoux, S. Marz, M. Payer, M. Pils, S. Rumler, H. Soukup, N. Wehn, P. Duzy, The Siemens high-level synthesis system CALLAS. *IEEE Trans. Very Large Scale Integr. Syst.* **1**(3), 244–253 (1993)
24. T. Filkorn, A method for symbolic verification of synchronous circuits, in *Proceedings of the Comp Hardware Descri Lang and Their Application (CHDL 91)*, pp. 229–239, Marseille, 1991
25. A. Kalavade, E. Lee, A hardware-software codesign methodology for DSP applications. *IEEE Des. Test Comput.* **10**(3), 16–28 (1993)
26. R. Ernst, J. Henkel, T. Benner, Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test Comput.* **10**(4), 64–75 (1993)
27. G. De Micheli, D. Ku, F. Mailhot, T. Truong, The Olympus synthesis system. *IEEE Des. Test Comput.* **7**(5), 37–53 (1990)
28. D. Thomas, J. Adams, H. Schmit, A model and methodology for hardware-software codesign. *IEEE Des. Test Comput.* **10**(3), 6–15 (1993)
29. C. Hoare, *Communicating Sequential Processes* (Prentice-Hall, Englewood Cliffs, NJ, 1985)
30. R. Gupta, G. De Micheli, Hardware-software cosynthesis for digital systems. *IEEE Des. Test Comput.* **10**(3), 29–41 (1993)
31. I. Bolsens, H. De Man, B. Lin, K. Van Rompaey, S. Vercateren, D. Verkest, Hardware/software co-design of digital telecommunication systems. *Proc. IEEE* **85**(3), 391–418 (1997)
32. D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, H. De Man, DSP specification using the SILAGE language, in *Proceedings of the International Conference on Acoustics Speech Signal Process*, pp. 1056–1060, Albuquerque, NM, 3–6 April 1990
33. P. Willekens et al., Algorithm specification in DSP station using data flow language. *DSP Appl.* **3**(1), 8–16 (1994)
34. N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous dataflow programming language Lustre. *Proc. IEEE* **79**(9), 1305–1320 (1991)
35. M. Van Canneyt, Specification, simulation and implementation of a GSM speech codec with DSP station. *DSP Multimed. Technol.* **3**(5), 6–15 (1994)
36. J. Buck, S. Ha, E. Lee, D. Messerschmitt, PTOLEMY: a framework for simulating and prototyping heterogeneous systems. *Int. J. Comput. Simul.* **4**, 1–34 (1992)
37. R. Lauwereins, M. Engels, M. Ade, J. Peperstraete, GRAPE-II: a system level prototyping environment for DSP applications. *IEEE Comput.* **28**(2), 35–43 (1995)
38. M. Rafie et al., Rapid design and prototyping of a direct sequence spread-spectrum ASIC over a wireless link. *DSP Multimed. Technol.* **3**(6), 6–12 (1994)
39. L. Semeria, K. Sato, G. De Micheli, Synthesis of hardware models in C with pointers and complex data structures. *IEEE Trans. VLSI Syst.* **9**(6), 743–756 (2001)
40. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, J. Hennessy, Suif: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIPLAN Notices* **28**(9), 67–70 (1994)
41. A. Kountouris, C. Wolinski, Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.* **7**(3), 380–412 (2002)
42. S. Gupta, R. Gupta, N. Dutt, A. Nikolau, Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.* **9**(4), 441–470 (2004)
43. W. Wang, A. Raghunathan, N. Jha, S. Dey, High-level synthesis of multi-process behavioral descriptions, in *Proceedings of the 16th IEEE International Conference on VLSI Design (VLSI'03)*, ISBN: 0-7695-1868-0, pp. 467–473, 4–8 January 2003
44. Z. Gu, J. Wang, R. Dick, H. Zhou, Incremental exploration of the combined physical and behavioral design space, in *Proceedings of the 42nd Annual Conference on Design. Automation DAC '05*, pp. 208–213, Anaheim, CA, 13–17 June 2005
45. L. Zhong, N. Jha, Interconnect-aware high-level synthesis for low power, in *Proceedings of the IEEE/ACM International Conference Computer-Aided Design*, ISBN: 0-7803-7607-2, pp. 110–117, November 2002
46. C. Huang, S. Ravi, A. Raghunathan, N. Jha, Generation of heterogeneous distributed architectures for memory-intensive applications through high-level synthesis. *IEEE Trans. Very Large Scale Integr.* **15**(11), 1191–1204 (2007)

47. K. Wakabayashi, C-based synthesis experiences with a behavior synthesizer, “Cyber”, in *Proceedings of the Design Automation and Test in Europe Conference*, ISBN: 0-7695-0078-1, pp. 390–393, Munich, 9–12 March 1999
48. W. Wang, T. Tan, J. Luo, Y. Fei, L. Shang, K. Vallerio, L. Zhong, A. Raghunathan, N. Jha, A comprehensive high-level synthesis system for control-flow intensive behaviors, in *Proceedings of the 13th ACM Great Lakes Symposium on VLSI GLSVLSI '03*, ISBN:1-58113-677-3, pp. 11–14, Washington, DC, 28–29 April 2003
49. B. Gal, E. Casseau, S. Huet, Dynamic memory access management for high-performance DSP applications using high-level synthesis. *IEEE Trans. Very Large Scale Integr.* **16**(11), 1454–1464 (2008)
50. S. Gupta, R. Gupta, N. Dutt, A. Nicolau, Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits, in *Proceedings of the IEEE Conference Computers and Digital Techniques*, ISSN: 1350–2387, vol. 150, no. 5, pp. 330–337, 22 September 2003
51. K. Wakabayashi, H. Tanaka, Global scheduling independent of control dependencies based on condition vectors, in *Proceedings of the 29th ACM/IEEE Design Automation Conference (DAC)*, ISBN: 0-8186-2822-7, pp. 112–115, Anaheim, CA, 8–12 June 1992
52. E. Martin, O. Santieys, J. Philippe, GAUT, an architecture synthesis tool for dedicated signal processors, in *Proceedings of the IEEE International European Design Automation Conference (Euro-DAC)*, pp. 14–19, Hamburg, September 1993
53. M. Molina, R. Ruiz-Sautua, P. Garcia-Repetto, R. Hermida, Frequent-pattern-guided multilevel decomposition of behavioral specifications. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **28**(1), 60–73 (2009)
54. K. Avnit, V. D’Silva, A. Sowmya, S. Ramesh, S. Parameswaran, Provably correct on-chip communication: a formal approach to automatic protocol converter synthesis. *ACM Trans. Des. Autom. Electron. Syst.* **14**(2), 19 (2009)
55. J. Keinert, M. Streubuhr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electron. Syst.* **14**(1), 1 (2009)
56. S. Kundu, S. Lerner, R. Gupta, Translation validation of high-level synthesis. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **29**(4), 566–579 (2010)
57. S. Paik, I. Shin, T. Kim, Y. Shin, HLS-l: a high-level synthesis framework for latch-based architectures. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **29**(5), 657–670 (2010)
58. A. Raghunathan, S. Dey, N. Jha, Register-transfer level estimation techniques for switching activity and power consumption, in *Digest of Technical Papers, International Conference on Computer-Aided Design (ICCAD)*, ISBN: 0-8186-7597-7, pp. 158–165, San Jose, CA, 10–14 November 1996
59. J. Rabaey, L. Guerra, R. Mehra, Design guidance in the power dimension, in *Proceedings of the 1995 International Conference on Acoustics, Speech, and Signal Processing*, ISBN: 0-7803-2431-5, pp. 2837–2840, Detroit, MI, 9–12 May 1995
60. R. Mehra, J. Rabaey, Exploiting regularity for low-power design, in *Digest of Technical Papers, International Conference on Computer-Aided Design (ICCAD)*, ISBN: 0-8186-7597-7, pp. 166–172, San Jose, CA, November 1996
61. A. Raghunathan, N. Jha, Behavioral synthesis for low power, in *Proceedings of the International Conference on Computer Design (ICCD)*, ISBN: 0-8186-6565-3, pp. 318–322, Cambridge, MA, 10–12 October 1994
62. L. Goodby, A. Orailoglu, P. Chau, Microarchitecture synthesis of performance-constrained low-power VLSI designs, in *Proceedings of the International Conference on Computer Design (ICCD)*, ISBN: 0-8186-6565-3, pp. 323–326, Cambridge, MA, 10–12 October 1994
63. E. Musoll, J. Cortadella, Scheduling and resource binding for low power, in *Proceedings of the Eighth Symposium on System Synthesis*, ISBN: 0-8186-7076-2, pp. 104–109, Cannes, 13–15 September 1995

64. N. Kumar, S. Katkoori, L. Rader, R. Vemuri, Profile-driven behavioral synthesis for low-power VLSI systems. *IEEE Des. Test Comput.* **12**(3), 70–84 (1995)
65. R. Martin, J. Knight, Power-profiler: optimizing ASICs power consumption at the behavioral level, in *Proceedings of the Design Automation Conference (DAC)*, ISBN: 0-89791-725-1, pp. 42–47, San Francisco, CA, 1995
66. I. Issenin, E. Brockmeyer, B. Durinck, N.D. Dutt, Data-reuse-driven energy-aware cosynthesis of scratch pad memory and hierarchical bus-based communication architecture for multiprocessor streaming applications. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **27**(8), 1439–1452 (2008)
67. M. Dossis, Intermediate Predicate Format for design automation tools. *J. Next Gener. Inform. Technol.* **1**(1), 100–117 (2010)
68. U. Nilsson, J. Maluszynski, *Logic Programming and Prolog*, 2nd edn. (John Wiley & Sons Ltd., Chichester, 1995)

Components and Services for IoT Platforms

Paving the Way for IoT Standards

Keramidas, G.; Voros, N.; Hübner, M. (Eds.)

2017, IX, 383 p. 128 illus., 106 illus. in color., Hardcover

ISBN: 978-3-319-42302-9